### 

### **Purpose**

The purpose of lab 5 was to obtain further experience with Verilog programming and overall familiarity of how to functionally verify a processor design. The first task included the drawing of diagrams regarding the single-cycle MIPS processor design. After building a provided DUT, the second task required verification of the testbench. The lab provided further practice with analyzing waveforms for further developing an understanding of MIPS instructions and functional verification.

**Approach**

For task 1, we were tasked with drawing the block diagrams for the following: datapath with microarchitecture details, control unit with microarchitecture, instruction memory and data memory, processor core, and the complete processor integrated with memory. The diagrams are depicted in Appendix A. To accomplish this task, we used the Vivado schematic tool, which generates a schematic for the given design, and the provided source code.

For the datapath with microarchitecture details, we spent a good deal of time making sure every wire was labeled accordingly using names in the given source code. All smaller modules that make up the microarchitecture were connected as described in the source code and Vivado schematic tool. The datapath diagram can be found in Appendix A, Figure A. We used the same approach with the control unit with microarchitecture. The control unit shows how the input opcode and function fields from a 32-bit instruction are used to generate the relevant control signals to control the datapath. The control unit diagram can be found in Appendix A, Figure B. Similarly, we used the provided resources to draw a diagram for the instruction and data memory modules. These modules show how input addresses are used to retrieve instructions and data from memory. These diagrams can be found in Appendix A, Figure C and Figure D. For the processor core, we made a diagram showing the interconnections between the control unit and datapath. These modules are also connected to any input and output signals relevant to the processor core. This diagram can be found in Appendix A, Figure E. Finally, our complete processor or top module was created using the previous diagrams. The diagram shows the connections between the memory and MIPS processor core. Additionally, the diagram shows which data signals appear on the simulation waveforms as well as any inputs to the system. This diagram can be found in Appendix A, Figure F.

For task 2, we were tasked with using the provided simulation source code to functionally verify the MIPS processor. The resulting waveforms are depicted in Appendix B. To accomplish this task, we made sure to include the given memory file in the project directory before running the simulation. After generating the waveforms with Vivado, we were able to see how the program counter incremented itself after each instruction was executed. The waveform also shows how each instruction generates an ALU output depending on the kind of instruction being executed. Additionally, the waveform shows the output for the second register source as this is used when writing to data memory. Note that no data is written to memory unless the we\_dm signal is high. Finally, the waveform shows the output of data memory which changes as an instruction directs the datapath and control unit requests something from memory. By looking at what an instruction should accomplish with the MARS tool, we can examine that the correct output is generated on the Vivado waveforms to functionally verify the MIPS processor. For example, for an add instruction the process is as follows: the instruction is loaded from memory using an address from the program counter, the relevant values are pulled from the register file using addresses given in the instruction, the computation is performed in the ALU, and the result is stored back to the register file using the address given in the instruction. This can all be verified with the output of the ALU in the waveform screenshots found in Appendix B. Other instructions are also visible in the waveform screenshots. Please see the descriptions of the waveforms in Appendix B for a detailed description of how waveforms were used to functionally verify the design of the single-cycle MIPS processor.

**Accomplished Tasks**

### Task 1: Drew the block diagrams based upon the provided code:

* Datapath (Figure A)
* Control Unit (Figure B)
* Instruction memory (Figure C)
* Data memory (Figure D)
* Processor core (Figure E)
* Complete processor (Figure F)

Task 2: Verification of the MIPS processor:

* The provided DUT was built
* The testbench for the MIPS processor was verified

**Conclusion**

The purpose of the lab was fulfilled. Further experience with Verilog programming was gained. From the first task, an understanding of MIPs processor design was developed. From the second task, verification of the testbench after building the DUT developed familiarity regarding the functionality of the processor. Additionally, a foundation regarding how to develop the techniques required for processor verification was built upon for future labs. With both tasks accomplished successfully, the goals of the lab were completed. Furthermore, a better understanding of how a single-cycle processor processes instructions was gained through functional verification of the processor. It will be interesting to see how this processor design evolves with pipelining and the addition of new instructions. In the next lab, we will attempt to run the processor off of the Basys3 board.

**Appendix A: Block Diagrams**

Table 1. Description of Block Diagrams

|  |  |
| --- | --- |
| **Module** | **Functionality** |
| Datapath | The datapath module is responsible for several things when processing an instruction. For example, the datapath increments the program counter register by 4 after every instruction. The datapath also performs arithmetic with the ALU component. Most importantly, the datapath hosts the register file wherein data is stored and retrieved as needed to perform operations on. A system of multiplexing control which data is put on each bus connecting the various modules. The datapath is made up of multiplexores, adders, a register file, an ALU, a sign extension module, and a single AND gate. |
| Control Unit | The control unit provides the control signals necessary for operation of the datapath. These include select signals for the multiplexors in the datapath, enable signals for the register file, signals to choose the operation of the arithmetic logic unit (ALU), and signals to writing to and reading from memory. The control unit is made up of two decoders: one for the main control signals for the datapath and one for the operation of the ALU. |
| Instruction Memory | The instruction memory holds all assembled MIPS instructions that were used to verify the function of the single-cycle MIPS processor. This memory unit functions as a read-only memory where the address to read from is controlled by the program counter. The output of this memory is a 32-bit instruction which is fed into the processor core for execution. |
| Data Memory | The data memory holds any data the user wants to store. For example, if a user wants to perform some simple arithmetic, they may want to save it to data memory for use later as values in the register file in the datapath are only temporary. This memory unit can be written to and read from as needed. |
| Processor Core | The processor core shows the control unit and datapath connections. It also includes any external signals that are fed into the processor core such as the clock, instructions, and reset signals. The core cannot alone process instructions. It must be connected to memory before it can operate. |
| Complete Processor | The complete processor is the processor core connected to the instruction and data memory modules. This enables the processor to automatically read instructions from the instruction memory, process the instructions, and store the result back to data memory as needed. |

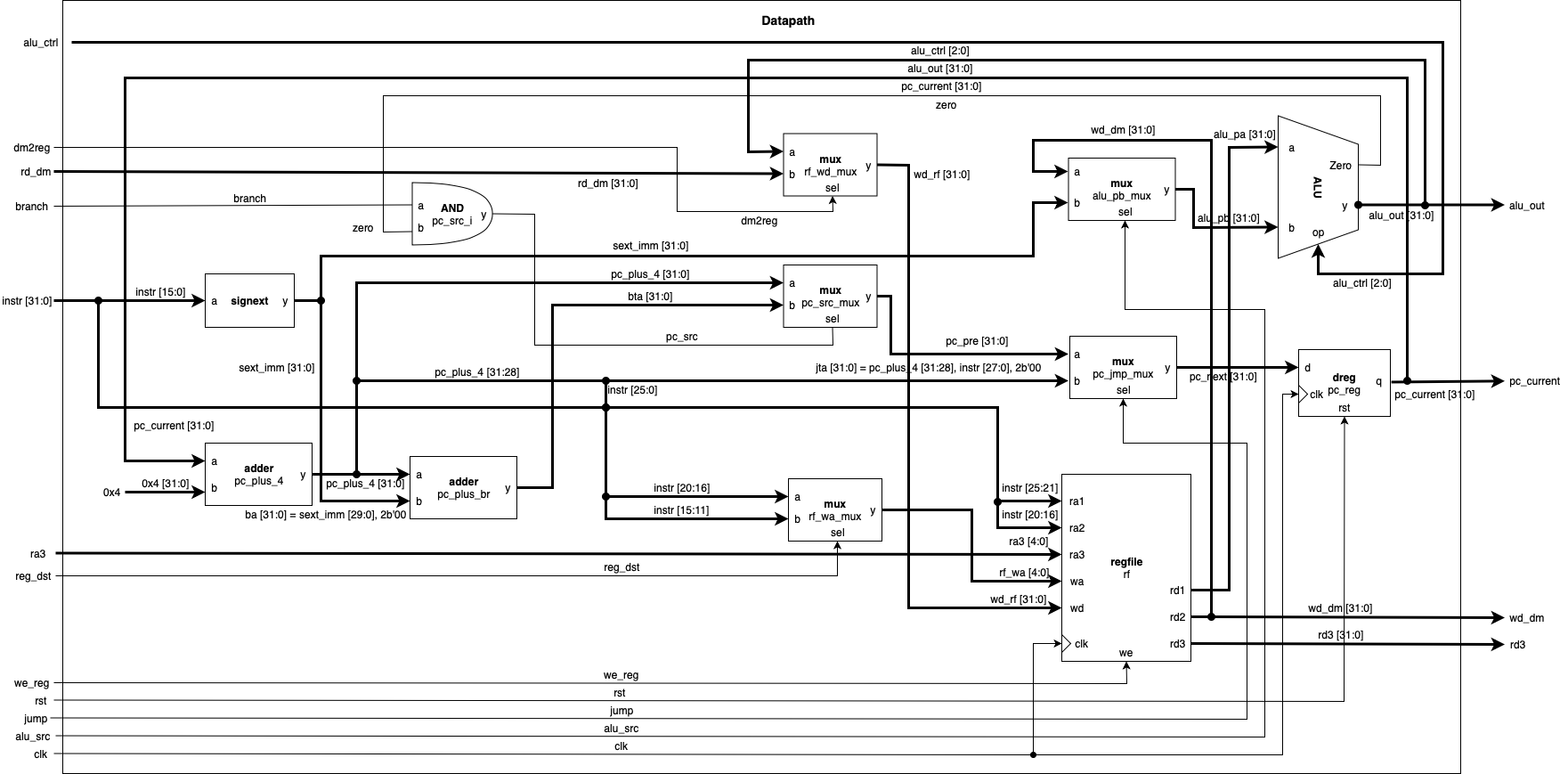
****

Figure A. Datapath

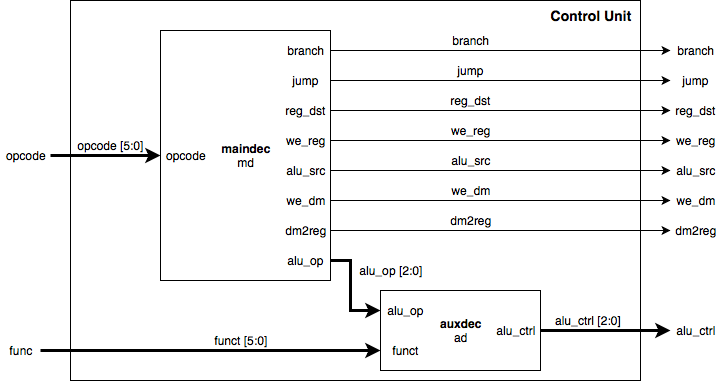
****

Figure B. Control Unit

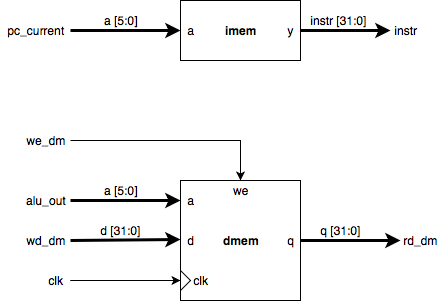


Figure C. IMEM

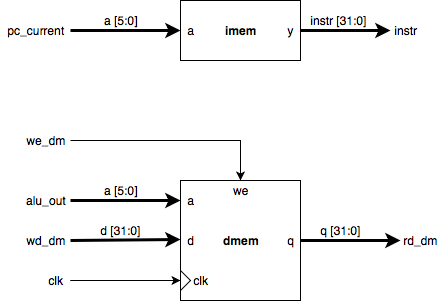


Figure D. DMEM

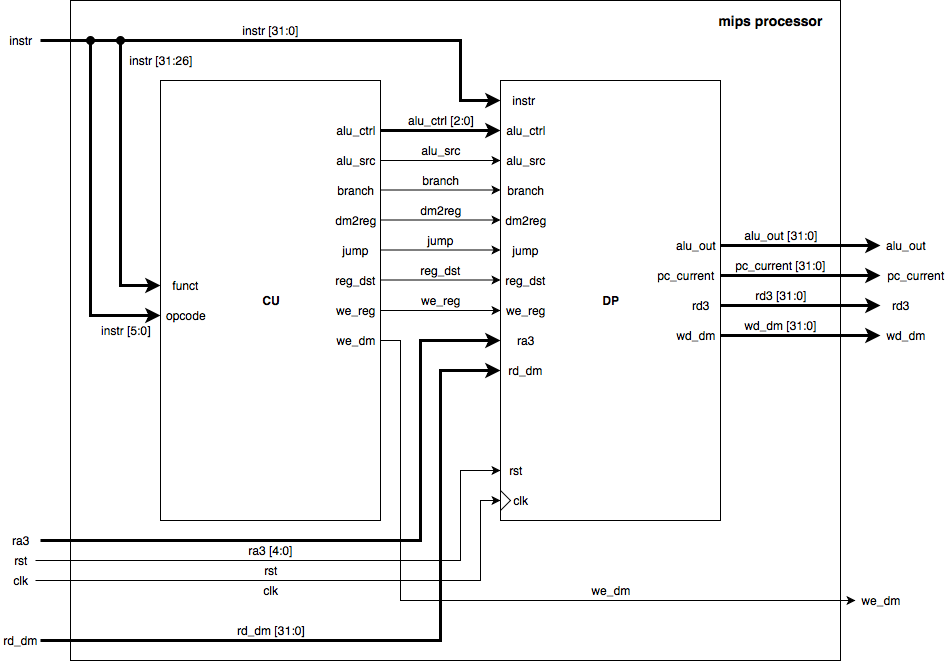


Figure E. Processor Core (CU-DP)

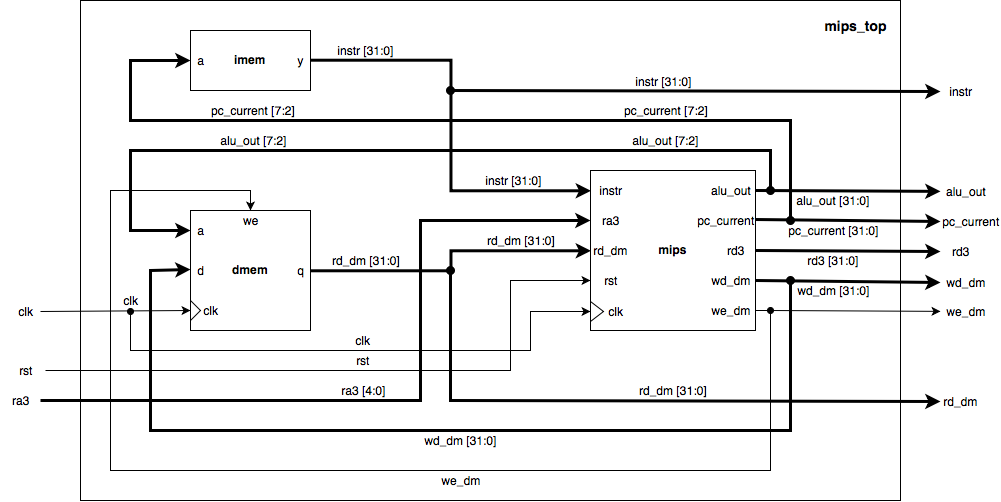
****

Figure F. Complete Processor

**Appendix B: Waveforms**

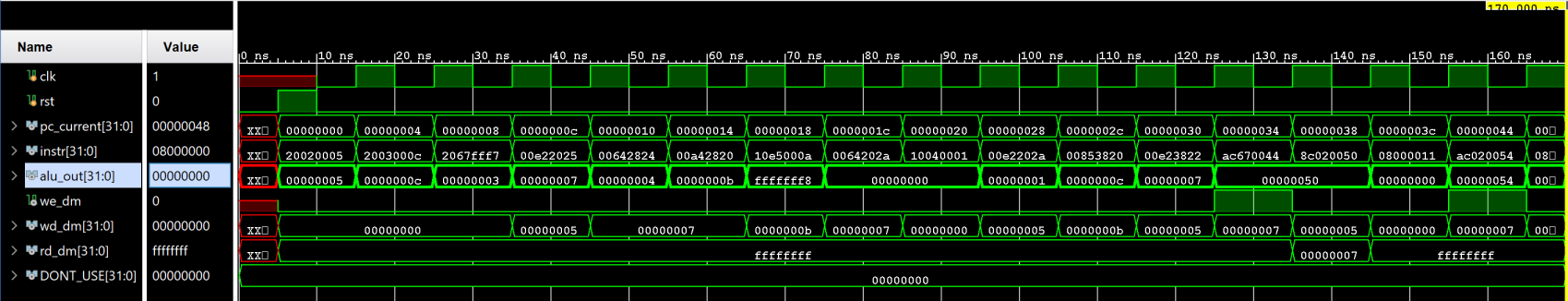


Figure G. This waveform screenshot shows the result of processing all instructions from the included *memfile.dat*. Signals shown include the clock, instruction, program counter, output of the arithmetic logic unit (ALU), write to data memory signal, and the output of the second source register in the register file. Note that we\_dm (write enable for data memory) is not high until the instruction 0xac670044 is executed. This is because a save word instruction is being processed to store data to data memory. Also note that each clock period contains exactly one instruction. This is correct for a single-cycle processor as the clock period is matched to the length of the time it takes to execute the most time-intensive instruction. See Figure H for a detailed description of how a single clock period is used to verify correct operation of the processor.

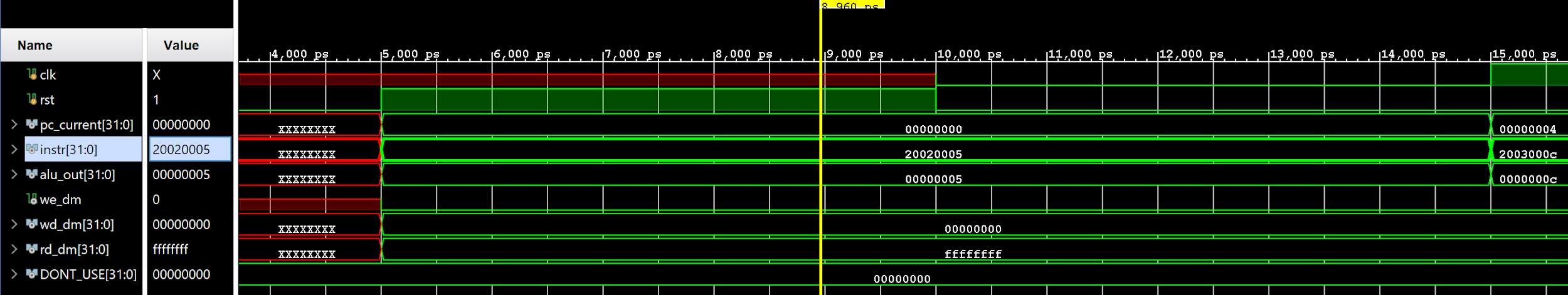


Figure H. A close up screenshot showing an I-Type instruction being executed by the processor. The signals shown are the same as in Figure G. The instruction shown is 0x2002005 (addi $2, $0, 5). For this instruction, the alu\_out signal shows 0x5. This is functionally correct as the value from $0 is added to the immediate value 5 in the ALU to generate 0x5. Notice that rd\_dm is 0xffffffff for this operation. This is because an I-Type instruction does not read data from memory (data memory module). Additionally, wd\_dm is 0x0 as wd\_dm corresponds to the output of the second source register in the register file of the datapath. Since an I-Type instruction only uses one source register and an immediate value encoded in the machine code, 0x0 is the correct value for wd\_dm. Finally, notice that the program counter shows 0x0 for this instruction. This is because this is the first instruction loaded from the instruction memory. The program counter is correctly incremented by 4 on the next clock cycle.

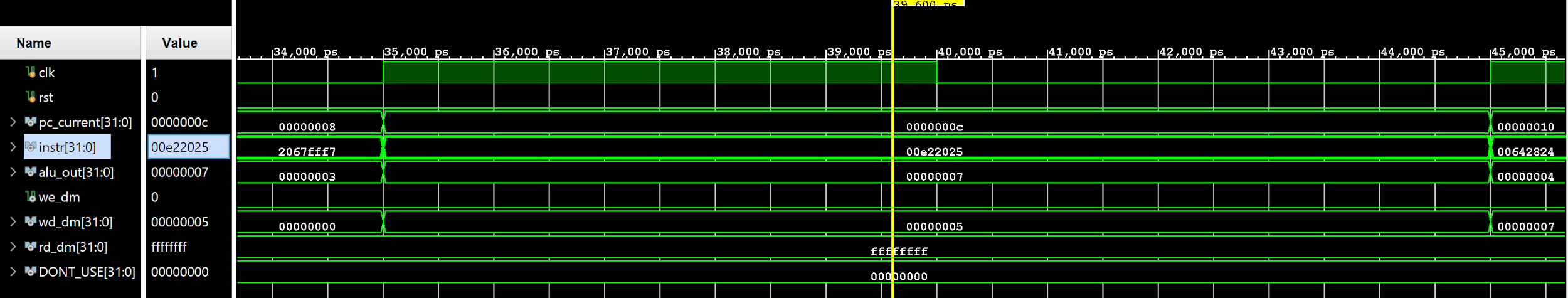
****

Figure I. This waveform screenshot shows an example of an R-Type instruction. The same signals are shown as in Figure G and H. For this instruction 0x00e22025 (or $4, $7, $2), the alu\_out shows 0x7 and wd\_dm shows 0x5. The value in $7 and $2 is 0x3 and 0x5, respectively. The bitwise-or of these two values results in 0x7 which matches the alu\_out for this instruction. wd\_dm (write data for data memory) shows 0x5 as this wire shows the output of the second source register given by the address in the instruction. $2 is the second source register in this example which contains 0x5. Therefore, this matches with the waveform screenshot. Note that wd\_dm is fed directly to the ALU in the datapath diagram in Figure A.

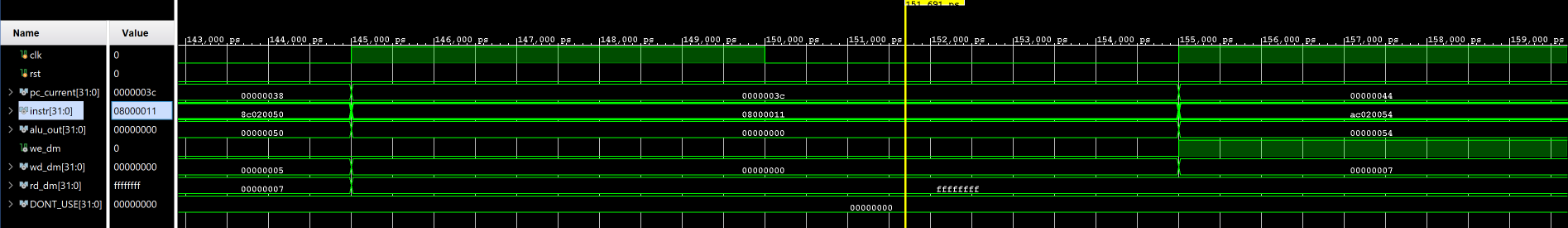
****

Figure J. This waveform screenshot shows a close up of a J-Type instruction. The same signals are shown as in the above waveform screenshots. The instruction being executed here is 0x08000011. The main thing to note in this screenshot is the transition in the program counter’s value between this instructions clock cycle and the next instruction. Because this is a jump instruction, the address of the next instruction to retrieve from instruction memory must be calculated. This can be found by concatenating the following bits: pc\_plus\_4[31:28] + instr[27:2] + 2’b00. This comes out to 0xac020054 which matches the program counter output of the next instruction.

**Appendix C: Source Code**

|  |
| --- |
| **auxdec.v** |
| module auxdec (  input wire [1:0] alu\_op,  input wire [5:0] funct,  output wire [2:0] alu\_ctrl  );  reg [2:0] ctrl;  assign {alu\_ctrl} = ctrl;  always @ (alu\_op, funct) begin  case (alu\_op)  2'b00: ctrl = 3'b010; // ADD  2'b01: ctrl = 3'b110; // SUB  default: case (funct)  6'b10\_0100: ctrl = 3'b000; // AND  6'b10\_0101: ctrl = 3'b001; // OR  6'b10\_0000: ctrl = 3'b010; // ADD  6'b10\_0010: ctrl = 3'b110; // SUB  6'b10\_1010: ctrl = 3'b111; // SLT  default: ctrl = 3'bxxx;  endcase  endcase  end  endmodule |

|  |
| --- |
| **controlunit.v** |
| module controlunit (  input wire [5:0] opcode,  input wire [5:0] funct,  output wire branch,  output wire jump,  output wire reg\_dst,  output wire we\_reg,  output wire alu\_src,  output wire we\_dm,  output wire dm2reg,  output wire [2:0] alu\_ctrl  );    wire [1:0] alu\_op;  maindec md (  .opcode (opcode),  .branch (branch),  .jump (jump),  .reg\_dst (reg\_dst),  .we\_reg (we\_reg),  .alu\_src (alu\_src),  .we\_dm (we\_dm),  .dm2reg (dm2reg),  .alu\_op (alu\_op)  );  auxdec ad (  .alu\_op (alu\_op),  .funct (funct),  .alu\_ctrl (alu\_ctrl)  );  endmodule |

|  |
| --- |
| **maindec.v** |
| module maindec (  input wire [5:0] opcode,  output wire branch,  output wire jump,  output wire reg\_dst,  output wire we\_reg,  output wire alu\_src,  output wire we\_dm,  output wire dm2reg,  output wire [1:0] alu\_op  );  reg [8:0] ctrl;  assign {branch, jump, reg\_dst, we\_reg, alu\_src, we\_dm, dm2reg, alu\_op} = ctrl;  always @ (opcode) begin  case (opcode)  6'b00\_0000: ctrl = 9'b0\_0\_1\_1\_0\_0\_0\_10; // R-type  6'b00\_1000: ctrl = 9'b0\_0\_0\_1\_1\_0\_0\_00; // ADDI  6'b00\_0100: ctrl = 9'b1\_0\_0\_0\_0\_0\_0\_01; // BEQ  6'b00\_0010: ctrl = 9'b0\_1\_0\_0\_0\_0\_0\_00; // J  6'b10\_1011: ctrl = 9'b0\_0\_0\_0\_1\_1\_0\_00; // SW  6'b10\_0011: ctrl = 9'b0\_0\_0\_1\_1\_0\_1\_00; // LW  default: ctrl = 9'bx\_x\_x\_x\_x\_x\_x\_xx;  endcase  end  endmodule |

|  |
| --- |
| **adder.v** |
| module adder (  input wire [31:0] a,  input wire [31:0] b,  output wire [31:0] y  );  assign y = a + b;    endmodule |

|  |
| --- |
| **alu.v** |
| module alu (  input wire [2:0] op,  input wire [31:0] a,  input wire [31:0] b,  output wire zero,  output reg [31:0] y  );  assign zero = (y == 0);  always @ (op, a, b) begin  case (op)  3'b000: y = a & b;  3'b001: y = a | b;  3'b010: y = a + b;  3'b110: y = a - b;  3'b111: y = (a < b) ? 1 : 0;  endcase  end  endmodule |

|  |
| --- |
| **datapath.v** |
| module datapath (  input wire clk,  input wire rst,  input wire branch,  input wire jump,  input wire reg\_dst,  input wire we\_reg,  input wire alu\_src,  input wire dm2reg,  input wire [2:0] alu\_ctrl,  input wire [4:0] ra3,  input wire [31:0] instr,  input wire [31:0] rd\_dm,  output wire [31:0] pc\_current,  output wire [31:0] alu\_out,  output wire [31:0] wd\_dm,  output wire [31:0] rd3  );  wire [4:0] rf\_wa;  wire pc\_src;  wire [31:0] pc\_plus4;  wire [31:0] pc\_pre;  wire [31:0] pc\_next;  wire [31:0] sext\_imm;  wire [31:0] ba;  wire [31:0] bta;  wire [31:0] jta;  wire [31:0] alu\_pa;  wire [31:0] alu\_pb;  wire [31:0] wd\_rf;  wire zero;    assign pc\_src = branch & zero;  assign ba = {sext\_imm[29:0], 2'b00};  assign jta = {pc\_plus4[31:28], instr[25:0], 2'b00};    // --- PC Logic --- //  dreg pc\_reg (  .clk (clk),  .rst (rst),  .d (pc\_next),  .q (pc\_current)  );  adder pc\_plus\_4 (  .a (pc\_current),  .b (32'd4),  .y (pc\_plus4)  );  adder pc\_plus\_br (  .a (pc\_plus4),  .b (ba),  .y (bta)  );  mux2 #(32) pc\_src\_mux (  .sel (pc\_src),  .a (pc\_plus4),  .b (bta),  .y (pc\_pre)  );  mux2 #(32) pc\_jmp\_mux (  .sel (jump),  .a (pc\_pre),  .b (jta),  .y (pc\_next)  );  // --- RF Logic --- //  mux2 #(5) rf\_wa\_mux (  .sel (reg\_dst),  .a (instr[20:16]),  .b (instr[15:11]),  .y (rf\_wa)  );  regfile rf (  .clk (clk),  .we (we\_reg),  .ra1 (instr[25:21]),  .ra2 (instr[20:16]),  .ra3 (ra3),  .wa (rf\_wa),  .wd (wd\_rf),  .rd1 (alu\_pa),  .rd2 (wd\_dm),  .rd3 (rd3)  );  signext se (  .a (instr[15:0]),  .y (sext\_imm)  );  // --- ALU Logic --- //  mux2 #(32) alu\_pb\_mux (  .sel (alu\_src),  .a (wd\_dm),  .b (sext\_imm),  .y (alu\_pb)  );  alu alu (  .op (alu\_ctrl),  .a (alu\_pa),  .b (alu\_pb),  .zero (zero),  .y (alu\_out)  );  // --- MEM Logic --- //  mux2 #(32) rf\_wd\_mux (  .sel (dm2reg),  .a (alu\_out),  .b (rd\_dm),  .y (wd\_rf)  );  endmodule |

|  |
| --- |
| **dreg.v** |
| module dreg # (parameter WIDTH = 32) (  input wire clk,  input wire rst,  input wire [WIDTH-1:0] d,  output reg [WIDTH-1:0] q  );  always @ (posedge clk, posedge rst) begin  if (rst) q <= 0;  else q <= d;  end  endmodule |

|  |
| --- |
| **mux2.v** |
| module mux2 #(parameter WIDTH = 8) (  input wire sel,  input wire [WIDTH-1:0] a,  input wire [WIDTH-1:0] b,  output wire [WIDTH-1:0] y  );  assign y = (sel) ? b : a;  endmodule |

|  |
| --- |
| **regfile.v** |
| module regfile (  input wire clk,  input wire we,  input wire [4:0] ra1,  input wire [4:0] ra2,  input wire [4:0] ra3,  input wire [4:0] wa,  input wire [31:0] wd,  output wire [31:0] rd1,  output wire [31:0] rd2,  output wire [31:0] rd3  );  reg [31:0] rf [0:31];  integer n;    initial begin  for (n = 0; n < 32; n = n + 1) rf[n] = 32'h0;  rf[29] = 32'h100; // Initialze $sp  end    always @ (posedge clk) begin  if (we) rf[wa] <= wd;  end  assign rd1 = (ra1 == 0) ? 0 : rf[ra1];  assign rd2 = (ra2 == 0) ? 0 : rf[ra2];  assign rd3 = (ra3 == 0) ? 0 : rf[ra3];  endmodule |

|  |
| --- |
| **signext.v** |
| module signext (  input wire [15:0] a,  output wire [31:0] y  );  assign y = {{16{a[15]}}, a};    endmodule |

|  |
| --- |
| **dmem.v** |
| module dmem (  input wire clk,  input wire we,  input wire [5:0] a,  input wire [31:0] d,  output wire [31:0] q  );  reg [31:0] ram [0:63];  integer n;  initial begin  for (n = 0; n < 64; n = n + 1) ram[n] = 32'hFFFFFFFF;  end  always @ (posedge clk) begin  if (we) ram[a] <= d;  end  assign q = ram[a];    endmodule |

|  |
| --- |
| **imem.v** |
| module imem (  input wire [5:0] a,  output wire [31:0] y  );  reg [31:0] rom [0:63];  initial begin  $readmemh ("memfile.dat", rom);  end  assign y = rom[a];    endmodule |

|  |
| --- |
| **mips\_top.v** |
| module mips\_top (  input wire clk,  input wire rst,  input wire [4:0] ra3,  output wire we\_dm,  output wire [31:0] pc\_current,  output wire [31:0] instr,  output wire [31:0] alu\_out,  output wire [31:0] wd\_dm,  output wire [31:0] rd\_dm,  output wire [31:0] rd3  );  wire [31:0] DONT\_USE;  mips mips (  .clk (clk),  .rst (rst),  .ra3 (ra3),  .instr (instr),  .rd\_dm (rd\_dm),  .we\_dm (we\_dm),  .pc\_current (pc\_current),  .alu\_out (alu\_out),  .wd\_dm (wd\_dm),  .rd3 (rd3)  );  imem imem (  .a (pc\_current[7:2]),  .y (instr)  );  dmem dmem (  .clk (clk),  .we (we\_dm),  .a (alu\_out[7:2]),  .d (wd\_dm),  .q (rd\_dm)  );  endmodule |

|  |
| --- |
| **mips.v** |
| module mips (  input wire clk,  input wire rst,  input wire [4:0] ra3,  input wire [31:0] instr,  input wire [31:0] rd\_dm,  output wire we\_dm,  output wire [31:0] pc\_current,  output wire [31:0] alu\_out,  output wire [31:0] wd\_dm,  output wire [31:0] rd3  );    wire branch;  wire jump;  wire reg\_dst;  wire we\_reg;  wire alu\_src;  wire dm2reg;  wire [2:0] alu\_ctrl;  datapath dp (  .clk (clk),  .rst (rst),  .branch (branch),  .jump (jump),  .reg\_dst (reg\_dst),  .we\_reg (we\_reg),  .alu\_src (alu\_src),  .dm2reg (dm2reg),  .alu\_ctrl (alu\_ctrl),  .ra3 (ra3),  .instr (instr),  .rd\_dm (rd\_dm),  .pc\_current (pc\_current),  .alu\_out (alu\_out),  .wd\_dm (wd\_dm),  .rd3 (rd3)  );  controlunit cu (  .opcode (instr[31:26]),  .funct (instr[5:0]),  .branch (branch),  .jump (jump),  .reg\_dst (reg\_dst),  .we\_reg (we\_reg),  .alu\_src (alu\_src),  .we\_dm (we\_dm),  .dm2reg (dm2reg),  .alu\_ctrl (alu\_ctrl)  );  endmodule |

|  |
| --- |
| **tb\_mips\_top.v** |
| module tb\_mips\_top;  reg clk;  reg rst;  wire we\_dm;  wire [31:0] pc\_current;  wire [31:0] instr;  wire [31:0] alu\_out;  wire [31:0] wd\_dm;  wire [31:0] rd\_dm;  wire [31:0] DONT\_USE;    mips\_top DUT (  .clk (clk),  .rst (rst),  .we\_dm (we\_dm),  .ra3 (5'h0),  .pc\_current (pc\_current),  .instr (instr),  .alu\_out (alu\_out),  .wd\_dm (wd\_dm),  .rd\_dm (rd\_dm),  .rd3 (DONT\_USE)  );    task tick;  begin  clk = 1'b0; #5;  clk = 1'b1; #5;  end  endtask  task reset;  begin  rst = 1'b0; #5;  rst = 1'b1; #5;  rst = 1'b0;  end  endtask    initial begin  reset;  while(pc\_current != 32'h48) tick;  $finish;  end  endmodule |

|  |
| --- |
| **memfile.dat** |
| 20020005  2003000C  2067FFF7  00E22025  00642824  00A42820  10E5000A  0064202A  10040001  20050000  00E2202A  00853820  00E23822  AC670044  8C020050  08000011  20020001  AC020054  08000000 |